

**Collaboard Authentication**

EXTERNAL AUTHENTICATION AND USER AUTHORIZATION

|  |  |
| --- | --- |
| Title: | Collaboard External Authentication and user authorization |
| Abstract: | This document describes the integration of external identity providers into Collaboard and user authorization |
| Status\* | Approved |
| Document Owner: | IBV |
| Document Author: |  |
| \*Draft, In progress, Reviewed, Authorized, Approved, Completed | |

|  |  |  |
| --- | --- | --- |
| **Document Control** | | |
| Document Distribution: | Internal: | All dev Teams |
|  | External: |  |
| Document Authorization: | Authorizer(s): |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Change History** | | | |
| Date | Ver | Changed By | Details |
| 18/05/2021 | 1.0 | D. Stellakis | Initial Draft |
| 19/05/2021 | 1.1 | G.P. Santopaolo | Document review |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Index

[Overview 3](#_Toc72326025)

[SAML 4](#_Toc72326026)

[Flows 4](#_Toc72326027)

[Bindings 4](#_Toc72326028)

[Assertions 4](#_Toc72326029)

[OAuth 2.0 7](#_Toc72326030)

[The Authorization Code Flow 7](#_Toc72326031)

[Get the User’s Permission 7](#_Toc72326032)

[Redirect Back to the Application 7](#_Toc72326033)

[Exchange the Authorization Code for an Access Token 8](#_Toc72326034)

[JSON Web Tokens 10](#_Toc72326035)

[JSON Web Token structure 10](#_Toc72326036)

[Header 10](#_Toc72326037)

[Payload 11](#_Toc72326038)

[Signature 11](#_Toc72326039)

[Putting it all together 11](#_Toc72326040)

[How do JSON Web Tokens work? 12](#_Toc72326041)

[Two-Factor Authentication 14](#_Toc72326042)

[Identifying a user 15](#_Toc72326043)

[Configuration 16](#_Toc72326044)

[Storage Authentication 17](#_Toc72326045)

[User Roles 18](#_Toc72326046)

[User Permissions 19](#_Toc72326047)

[References 20](#_Toc72326048)

# Overview

The purpose of this document is to provide information about the way the Collaboard authentication works.

The authentication/authorization process involves sending from the client to the webserver sensitive information, such as usernames, passwords, and tokens. Therefore, the client-server communication must be done over HTTPS.

For external authentication, the user is authenticated in an external authentication service provided by the customer. No passwords are entered or kept in the Collaboard database. Collaboard supports two different protocols for external authentication, SAML, and OAuth.

# SAML

SAML stands for Security Assertion Markup Language. It is an XML-based open-standard for transferring identity data between two parties: an identity provider (IdP) and a service provider (SP). We are compatible with identity providers supporting the SAML 2.0 HTTP Redirect / POST Binding.

* Identity Provider — Performs authentication and passes the user's identity and authorization level to the service provider.
* Service Provider — Trusts the identity provider and authorizes the given user to access the requested resource.

## Flows

SAML supports two different types of flows: those initiated by the service provider and those initiated by the identity provider. In our case, we cover the SP-initiated flow. In SP-initiated flows, you start at the service provider (IBV), are redirected to the identity provider to authenticate, and are then redirected back to the service provider. This flow is usually initiated when a user clicks on the "Login with SSO" button or something similar.

## Bindings

Bindings are the format in which data is transferred between service providers and identity providers. The two most popular are HTTP Redirect Binding and HTTP POST Binding. HTTP Redirect Bindings transfer data using HTTP redirects and query parameters; this type of binding is typically used in authentication requests. HTTP POST Binding transfer data using HTTP POST forms, this type of binding is generally used in authentication responses.

## Assertions

Assertions are statements made by the identity provider about the principal. For example, the principal's email address and groups/roles the principal may be associated with. Assertions are used by the service provider to create and configure sessions for a principal.

A typical sign-in flow using SAML redirect binding is:

Diagram

Description automatically generated

To integrate with an identity provider using the SAML 2.0 protocol, we will need the XML Metadata of the provider. A sample IdP XML metadata file would look like this:

<EntityDescriptor entityID="urn:idp.example.org" xmlns="urn:oasis:names:tc:SAML:2.0:metadata">

<IDPSSODescriptor protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">

<KeyDescriptor use="signing">

<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">

<X509Data>

<X509Certificate>MIIDBzCCAe+gAwIBAgI...P3Z3TTTs=</X509Certificate>

</X509Data>

</KeyInfo>

</KeyDescriptor>

<SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect" Location="https://idp.example.org/saml/logout"/>

<SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="https://idp.example.org/saml/logout"/>

<NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress</NameIDFormat>

<NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:persistent</NameIDFormat>

<NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:transient</NameIDFormat>

<SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect" Location="https://idp.example.org/saml"/>

<SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="https://idp.example.org/saml"/>

<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri" FriendlyName="E-Mail Address" xmlns="urn:oasis:names:tc:SAML:2.0:assertion"/>

<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri" FriendlyName="Given Name" xmlns="urn:oasis:names:tc:SAML:2.0:assertion"/>

<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri" FriendlyName="Name" xmlns="urn:oasis:names:tc:SAML:2.0:assertion"/>

<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri" FriendlyName="Surname" xmlns="urn:oasis:names:tc:SAML:2.0:assertion"/>

<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri" FriendlyName="Name ID" xmlns="urn:oasis:names:tc:SAML:2.0:assertion"/>

</IDPSSODescriptor>

</EntityDescriptor>

Alternatively, we would need the following information from the identity provider:

* IdP EntityID
* IdP Login URL
* IdP Logout URL
* IdP public certificate
* NameId Format
* User-specific attributes returned by the authentication response.

# OAuth 2.0

We support the OAuth 2.0 protocol for SSO. Currently, we have integration with Google, Microsoft, Apple, and Active Directory (ADFS) SSO identity providers.

An external login provider needs to support OAuth2 + OpenID to be used for external user authentication. Our service API is using the Authorization Code flow.

## The Authorization Code Flow

The Authorization Code grant type is used by web and mobile apps. At a high level, the flow has the following steps:

* The application opens a browser to send the user to the OAuth server.
* The user sees the authorization prompt and approves the app's request.
* The user is redirected back to the application with an authorization code in the query string.
* The application exchanges the authorization code for an access token.

## Get the User's Permission

OAuth is all about enabling users to grant limited access to applications. The application first needs to decide which permissions it is requesting, then send the user to a browser to get their permission. To begin the authorization flow, the application constructs a URL like the following and opens a browser to that URL.

https://authorization-server.com/auth ?response\_type=code

&client\_id=12345

&redirect\_uri=https%3A%2F%2Fexample-app.com%2Fcallback

&scope=email

Here is each query parameter explained:

* response\_type=code - This tells the authorization server that the application is initiating the authorization code flow.
* client\_id - The public identifier for the application, obtained when the developer first registered the application.
* redirect\_uri - Tells the authorization server where to send the user back to after they approve the request.
* scope - One or more space-separated strings indicating which permissions the application is requesting. The specific OAuth API you are using will define the scopes that it supports.

When the user visits this URL, the authorization server will present them with a prompt asking if they would like to authorize this application's request.

## Redirect Back to the Application

If the user approves the request, the authorization server will redirect the browser back to the redirect\_uri specified by the application, adding a code and state to the query string.

For example, the user will be redirected back to a URL such as

https://example-app.com/redirect

?code=g0ZGZmNjVmOWIjNTk2NTk4ZTYyZGI3

The code is the authorization code generated by the authorization server. This code is relatively short-lived, typically lasting between 1 to 10 minutes, depending on the OAuth service.

## Exchange the Authorization Code for an Access Token

Now that the application has the authorization code, it can use that to get an access token.

The application makes a POST request to the service's token endpoint with the following parameters:

* grant\_type=authorization\_code - This tells the token endpoint that the application is using the Authorization Code grant type.
* code - The application includes the authorization code it was given in the redirect.
* redirect\_uri - The same redirect URI that was used when requesting the code. Some APIs do not require this parameter, so you'll need to double-check the documentation of the particular API you're accessing.
* client\_id - The application's client ID.
* client\_secret - The application's client secret. This ensures that the request to get the access token is made only from the application and not from a potential attacker that may have intercepted the authorization code.

The token endpoint will verify all the parameters in the request, ensuring the code has not expired and that the client ID and secret match. If everything checks out, it will generate an access token and return it in the response:

HTTP/1.1 200 OK

Content-Type: application/json

Cache-Control: no-store

Pragma: no-cache

{

"access\_token":"MTQ0NjJkZmQ5OTM2NDE1ZTZjNGZmZjI3",

"token\_type":"bearer",

"expires\_in":3600,

"refresh\_token":"IwOGYzYTlmM2YxOTQ5MGE3YmNmMDFkNTVk",

"scope":"email"

}

The code exchange step ensures that an attacker is not able to intercept the access token, since the access token is always sent via a secure backchannel between the application and the OAuth server.

# JSON Web Tokens

After authenticating with an external provider, either via SAML or OAuth, the server will return to the calling client application a Collaboard-specific authorization token. This token will be used in every communication from the client to the server, being web API calls or SignalR calls, and the server will validate it every time, to allow or disallow access to the called API. The token is a JSON Web Token.

JSON Web Token (JWT) is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.

Although JWTs can also be encrypted to also provide secrecy between parties, we use signed tokens. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

Here are some scenarios where JSON Web Tokens are useful:

* **Authorization**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.
* **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content has not been tampered with.

## JSON Web Token structure

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

* Header
* Payload
* Signature

### Header

The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

For example:

{

"alg": "HS256",

"typ": "JWT"

}

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

### Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.

[**Registered claims**](https://tools.ietf.org/html/rfc7519#section-4.1): These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub** (subject), **aud** (audience), and others.

Notice that the claim names are only three characters long as JWT is meant to be compact.

[**Public claims**](https://tools.ietf.org/html/rfc7519#section-4.2): These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the [IANA JSON Web Token Registry](https://www.iana.org/assignments/jwt/jwt.xhtml) or be defined as a URI that contains a collision resistant namespace.

[**Private claims**](https://tools.ietf.org/html/rfc7519#section-4.3): These are the custom claims created to share information between parties that agree on using them and are neither registered or public claims.

An example payload could be:

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

### Signature

To create the signature part, you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example, if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)

The signature is used to verify the message was not changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

### Putting it all together

The output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.

The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.



## How do JSON Web Tokens work?

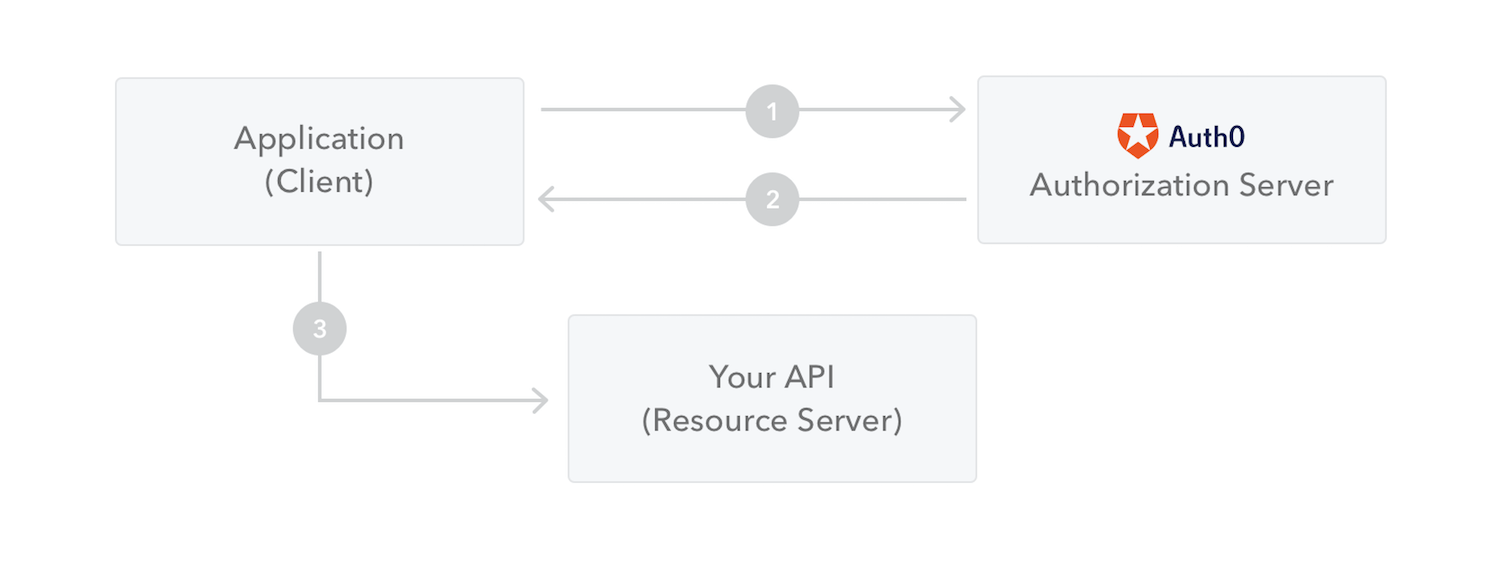
In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned. Since tokens are credentials, great care must be taken to prevent security issues. In general, you should not keep tokens longer than required.

Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the **Authorization** header using the **Bearer** schema. The content of the header should look like the following:

Authorization: Bearer <token>

This can be, in certain cases, a stateless authorization mechanism. The server's protected routes will check for a valid JWT in the Authorization header, and if it is present, the user will be allowed to access protected resources. If the JWT contains the necessary data, the need to query the database for certain operations may be reduced, though this may not always be the case.

If the token is sent in the Authorization header, Cross-Origin Resource Sharing (CORS) will not be an issue as it does not use cookies.

The following diagram shows how a JWT is obtained and used to access APIs or resources:

1. The application or client requests authorization to the authorization server. This is performed through one of the different authorization flows. For example, a typical [OpenID Connect](http://openid.net/connect/) compliant web application will go through the /oauth/authorize endpoint using the [authorization code flow](http://openid.net/specs/openid-connect-core-1_0.html#CodeFlowAuth).
2. When the authorization is granted, the authorization server returns an access token to the application.
3. The application uses the access token to access a protected resource (like an API).

# Two-Factor Authentication

For added security, users can choose to have Two-Factor Authentication (2FA) enabled for their account. 2FA means that the user must provide a one-time password (OTP) that he has obtained using a client authenticator app (such as Google Authenticator or Microsoft Authenticator) or sent to him via email from the web API.

An OTP, also known as a one-time pin or dynamic password, is a valid password for only one login session or transaction on a computer system or other digital device. OTPs avoid many shortcomings that are associated with traditional (static) password-based authentication. Many implementations also incorporate two-factor authentication by ensuring that the one-time password requires access to something a person has (such as a small keyring fob device with the OTP calculator built into it, or a smartcard or specific cellphone) as well as something a person knows (such as a PIN).

The most crucial advantage that OTPs address is that they are not vulnerable to replay attacks in contrast to static passwords. This means that a potential intruder who manages to record an OTP that was already used for logging in to a service or conducting a transaction will not be able to abuse it since it will no longer be valid. A second significant advantage is that a user who uses the same (or similar) password for multiple systems is not made vulnerable on all of them if an attacker gains the password for one of these. Many OTP systems also aim to ensure that a session cannot easily be intercepted or impersonated without knowledge of unpredictable data created during the previous session, thus reducing the attack surface further.

# Identifying a user

Once a user uses an identity provider for signing in to Collaboard, the system will use his user principal name (upn) returned by the SAML identity server, as well as other claims returned (such as first name, last name, email), to create the user's profile in Collaboard. The user will need to review his profile and accept the terms and conditions of the Collaboard application to proceed to use Collaboard.

If the user's profile is already present in the system, it will be located and reused, provided that the user has accepted the terms of service.

Since Collaboard does not keep any private user data in its data storage, all that it needs to identify a user is the upn claim returned by the SAML server, matching it to its username field. Thus, we can create user profiles for external users into the Collaboard data storage having only the username as a required field. When a user chooses to authenticate externally via SAML, his profile will be completed after successful authentication.

# Configuration

For the external provider authentication to work, each application that wishes to use it shall be assigned an app code to uniquely identify the app and for the server to load the configuration settings for this app. For instance, if the client web app wishes to use the external provider authentication functionality, we may assign it the code react-web, and it shall pass it onto each web API call. The setting is provided as a JSON object (serialized as a string) in the configuration settings, similar to the following:

Text

Description automatically generated

The configuration setting is defined as array of apps, and in each app there is an array of login providers with their corresponding settings. The provider properties are:

|  |  |  |
| --- | --- | --- |
| Property | Data Type | Meaning |
| Provider | String | The provider's name |
| ClientId | String | The client ID of the provider's app |
| ClientSecret | String | The secret key / certificate of the provider's app |
| AuthorizeUri | String | The authorization uri of the provider |
| TokenUri | String | The token uri of the provider |
| RedirectUri | String | The redirect uri after a successful provider login |
| Scope | String | The authorization scope for the provider |
| TFAEnabled | Boolean | True to use two-factor authentication for this provider, if the user has TFA enabled in his profile. |

The server API will use this information to integrate with the external login provider specified.

# Storage Authentication

The Collaboard application, apart from the Web API and the SignalR services, also uses storage to store all the files, photos and videos uploaded by users for each project.

To access the storage, each user needs to have a token that is valid only for this user and that project. This token is generated by calling a specific server API, which in turn, after validating the user's permissions for the project, returns a token that needs to be included in the storage request to access the files.

Suppose the Collaboard installation is using an external storage service, such as Azure Blob storage or Amazon S3. In that case, the storage token is generated using the storage service APIs, and the web client will need to include it in every request to these APIs.

Suppose the Collaboard installation is using the internal MFT storage. In that case, the storage token is generated using the internal MFT library APIs, and it will also need to be included in all MFT requests. The token is encrypted and short-lived for maximum security.

# User Roles

Users can have roles, and these roles determine their access levels for the operations and data of the system. If a user does not have a role, it is assumed that he is a simple user.

If needed, the roles of a logged-in user can be derived from the authorization token, from the payload section after JWT decoding, in the property roles that is a string array including all the different roles of the user:

{

"sub": "test@test.gr",

"exp": 1581966673,

"roles": ["UserManager"]

}

Each time a web API method is called, the server will check if the user has the necessary role to invoke it, as defined in the method's signature. If not, an HTTP 401 Status will be returned.

Currently, the following roles are defined in the Collaboard backend application:

|  |  |
| --- | --- |
| Role | Definition |
| UserManager | Has access to call the auth management backend API |
| LicensingManager | Has access to call the licensing management backend API, in regard to managing team members |
| CollaboardManager | Has access to call the Collaboard management backend API |

Using the backend application, a user that has the UserManager role can add or remove roles from other users, as needed.

# User Permissions

After successfully authenticating using an external identity provider, a user can use the Collaboard application to create and access projects. For each project, the user is given an access permission applicable to that project only. Based on these permissions, the system will allow or deny the user to do specific tasks for that project.

Currently, the available set of allowed permissions are:

* Read – The user has only access to view the project.
* Read/Write – The user has access to view and modify the project and can invite other users.
* Facilitator – Has all the read/write permissions, and in addition he can use the project presentation features.
* Owner – The user has full access to the project.

When a user invites another user to join a project, he will select the level of permissions the new user will have in that project.

In addition, using the backend functionality, an administrator can change the permission level for a user in a project, transfer the project ownership to another user, or remove a user from a project.

# References

* Microsoft Docs - Shared Access Signatures: Understanding the SAS Model  
  <https://docs.microsoft.com/en-us/azure/storage/common/storage-sas-overview>
* Wikipedia - Security Assertion Markup Language  
  <https://en.wikipedia.org/wiki/Security_Assertion_Markup_Language>
* <https://goteleport.com/blog/how-saml-authentication-works/>
* <https://developer.okta.com/blog/2018/04/10/oauth-authorization-code-grant-type>
* <https://jwt.io/>